Java Database Connectivity…

- Programs written in Java are able to communicate with relational databases (whether local or remote) via the Java Database Connectivity (JDBC) API

- In order to use JDBC for the accessing of data from a particular type of relational database, it is necessary to provide some mediating software that will allow JDBC to communicate with the vendor-specific API for that database

- Such software is referred to as a ***driver***

- Suitable drivers are usually supplied either by the database vendors themselves or by third parties

- ODBC drivers were originally available only for Microsoft (MS) databases, other vendors and third party suppliers have since brought out ODBC drivers for most of the major non-MS databases

- The most commonly used version of JDBC is currently JDBC 2, though there are still plenty of JDBC 1 drivers around, as well as an increasing number of JDBC 3 drivers

Using JDBC to access a database requires several steps

1. Load the database driver.

2. Establish a connection to the database.

3. Use the connection to create a *Statement* object and store a reference to this object

4. Use the above *Statement* reference to run a specific query or update statement and accept the result(s)

5. Manipulate and display the results (if a query) or check/show number of database rows affected (for an update)

6. Repeat steps 4 and 5 as many times as required for further queries/updates.

7. Close the connection.

Load the Database Driver

- This is achieved via static method ***forName*** of class *Class*

```
Class.forName("com.mysql.jdbc.Driver")
```

Establish a Connection to the Database

- We declare a ***Connection*** reference and call static method ***getConnection*** of class ***DriverManager*** to return a ***Connection*** object for this reference. Method ***getConnection*** takes three *String* arguments:
- a URL-style address for the database
- a user name;
- a password.

Example:

```
Connection con =
    DriverManager.getConnection("jdbc:mysql://
     localhost/database1","root", "");
```

Create a Statement Object and Store its Reference

- A *Statement* object is created by calling the ***createStatement*** method of our *Connection* object

- The address of the object returned by this call to *createStatement* is saved in a *Statement* reference

Example:

```
Statement stm = con.createStatement ( );
```

Run a Query/Update and Accept the Result(s)

- DML (Data Manipulation Language) statements in SQL may be divided into two categories:

- Those that retrieve data from a database (i.e., SELECT statements)

and

- Those that change the contents of the database in some way (INSERT, DELETE and UPDATE statements)

- Class *Statement* has methods ***executeQuery*** and ***executeUpdate*** that are used to execute these two categories respectively

- The former method returns a ***ResultSet*** object, while the latter returns an integer that indicates the number of database rows that have been affected by the updating operation

- It is common practice to store the SQL query in a ***String*** variable and then invoke ***executeQuery*** with this string as an argument, in order to avoid a rather cumbersome invocation line

Example 1:

```
String selectAll = "SELECT * FROM Accounts";
ResultSet results =stm.executeQuery(selectAll);
```

Example 2:

```
String selectFields = "SELECT acctNum, balance FROM
    Accounts";
ResultSet results = stm.executeQuery(selectFields);
```

Manipulate/Display/Check Result(s)

- The *ResultSet* object returned in response to a call of *executeQuery* contains the database rows that satisfy the query's search criteria

- Having moved to the particular row of interest via any of the methods, we can retrieve data via either the field name or the field position

- int getInt (String <columnName>)
- int getInt (int <columnIndex>)
- String getString (String <columnName>)
- String getString (int <columnIndex>)

- Initially, the *ResultSet* cursor/pointer is positioned **before** the first row of the query results, so method *next* must be called before attempting to access the results

- Such rows are commonly processed via a while loop that checks the Boolean return value of this method first (to determine whether there is any data at the selected position)

Example Table…

```
String select = "SELECT * FROM Accounts";
ResultSet results = stm.executeQuery(select);
while (results.next())
    {
    System.out.println("Account no.“ + results.getInt(1));

    System.out.println("Account holder: “ + results.getString(3) +
    " “ + results.getString(2));

    System.out.println("Balance: “ + results.getFloat(4));

    System.out.println ();
    }
```

NOTE: Column/field numbers start at 1, not 0

- Alternatively, column/field names can be used

For example:

```
System.out.println("Account no.“ +
results.getInt("acctNum"));
```

Close the Connection

- This is achieved by calling method *close* of our *Connection* object and should be carried out as soon as the processing of the database has finished

```
con.close();
```

- Statement objects may also be closed explicitly via the identically-named method of

  our *Statement* object.

  For example:

```
stm.close();
```

Modifying the Database Contents

- INSERT, DELETE and UPDATE statements
- We shall have to submit our SQL statements via the *executeUpdate* method

Example 1:

```
String s1 = "INSERT INTO Accounts“ + " VALUES (123456,'Smith',“ + "'John James',752.85)";

int result = stmt.executeUpdate(s1);
```

Example 2:

```
String s2 = "UPDATE Accounts“ + " SET surname = 'Bloggs',“ + "firstNames = 'Fred Joseph'“ + " WHERE acctNum = 123456";
stmt.executeUpdate(s2);
```

Example 3:

```
String s3 = "DELETE FROM Accounts“
    + " WHERE balance < 100";

result = stmt.executeUpdate(s3);
```

```
int result = statement.executeUpdate(insert);
if (result==0)
    System.out.println("* Insertion failed! *");
```

Transactions…

- A transaction is one or more SQL statements that may be grouped together as a single processing entity

- If only some of the statements are executed, then the database is likely to be left in an inconsistent state

- The SQL statements used to implement transaction processing are **COMMIT** and **ROLLBACK**, which are mirrored in Java by the *Connection* interface methods ***commit*** and ***rollback***

- ***commit*** is used at the end of a transaction to commit/finalize the database changes

- ***rollback*** is used (in an error situation) to restore the database to the state it was in prior to the current transaction (by undoing any statements that may have been executed)

- By default, however, JDBC automatically commits each individual SQL statement that is applied to a database

- In order to change this default behavior so that transaction processing may be carried out, we must first execute ***Connection*** method ***setAutoCommit*** with an argument of false (to switch off auto-commit).

```
..............................
con.setAutoCommit(false);
..............................
try
{
//Assumes existence of 3 SQL update strings
//called update1, update2 and update3.
stmt.executeUpdate(update1);
stmt.executeUpdate(update2);
stmt.executeUpdate(update3);
con.commit();
}
catch(SQLException sqlEx)
{
con.rollback();
System.out.println("* SQL error! Changes aborted... *");
}
..............................
```

Meta Data…

- Meta data is 'data about data'

- There are two categories of meta data available through the JDBC API:

  • Data about the rows and columns returned by a query (i.e., data about *ResultSet* objects);

  • Data about the database as a whole.

- The first of these is provided by interface ***ResultSetMetaData***, an object of which is returned by the *ResultSet* method ***getMetaData***

- Information available from a *ResultSetMetaData* object includes the following:
- the number of fields/columns in a *ResultSet* object;
- the name of a specified field;
- the data type of a field;
- the maximum width of a field;
- the table to which a field belongs.

- Data about the database as a whole is provided by interface ***DatabaseMetaData***, an object of which is returned by the *Connection* method ***getMetaData***

- However, most Java developers will rarely find a need for ***DatabaseMetaData***

- int getColumnCount()
- String getColumnName(<colNumber>)
- String getColumnTypeName(<colNumber>)

Scrollable *ResultSets* in JDBC 2…

- In all our examples so far, movement through a *ResultSet object has been confined* to the forward direction only, and even that has been restricted to moving by one row at a time

- With the emergence of JDBC 2 in Java 2, however, a great deal more flexibility was made available to Java programmers by the introduction of the following *ResultSet methods:*

boolean first()

boolean last()

boolean previous()

boolean relative (int <rows>)

boolean absolute(int <rows>)

For the first 3 methods, as with method *next, the return value in each case indicates whether or not there is* data at the specified position

- Method ***relative*** *takes a signed argument and moves forwards/backwards the* specified number of rows

For example:

```
results.relative(-3);        //Move back 3 rows.
```

- Method ***absolute*** *also takes a signed argument and moves to the specified absolute* position, counting either from the start of the *ResultSet (for a positive argument) or* from the end of the *ResultSet (for a negative argument)*

For example:

```
results.absolute(3);
	//Move to row 3 (from start of ResultSet).
```

- Before any of these new methods can be employed, however, it is necessary to create a *scrollable ResultSet*

- This is achieved by using an overloaded form of the Connection method *createStatement* that takes two integer arguments

```
Statement createStatement(int <resultSetType>, int
<resultSetConcurrency>)
```

- There are three possible values that the first argument can take to specify the type of *ResultSet object that is to be created*

- *These three values are identified by the* following static constants in interface *ResultSet:*

```
TYPE_FORWARD_ONLY
TYPE_SCROLL_INSENSITIVE
TYPE_SCROLL_SENSITIVE
```

- The first option allows only forward movement through the *ResultSet*

- The second and third options allow movement of the *ResultSet's cursor* both forwards and backwards through the rows

- The difference between these two is that *TYPE_SCROLL_SENSITIVE causes any changes made to the data rows to be* reflected dynamically in the *ResultSet object, while TYPE_SCROLL_INSENSITIVE* does not

- There are two possible values that the second argument to *createStatement can* take

- These are identified by the following static constants in interface *ResultSet:*

  CONCUR_READ_ONLY

  CONCUR_UPDATABLE

- The first means that we cannot make changes to the *ResultSet rows, while the second will allow changes to be made*

Modifying Databases via Java Methods…

Another very useful feature of JDBC 2 is the ability to modify ResultSet rows directly via Java methods (rather than having to send SQL statements), and to have those changes reflected in the database itself

- In order to do this, it is necessary to use the second version of *createStatement again (i.e., the version that takes two* integer arguments) and supply *ResultSet.CONCUR_UPDATABLE as the second* argument

- The updateable *ResultSet object does not have to be scrollable, but, when* making changes to a *ResultSet, we often want to move freely around the ResultSet* rows, so it seems sensible to make the *ResultSet scrollable*

Example

```
Statement stm = con.createStatement(
ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

As usual, there are three types of change that we can carry out on the data in a database:

- updates (of some/all fields of a selected row)
- insertions (of new data rows)
- deletions (of existing database rows)

- There is a set of ***update**** *methods* (analogous to the *get*** methods that we use to retrieve the data from a row within a ResultSet),* each of these methods corresponding to one of the data types that may be held in the database

- For example, there are methods ***updateString*** *and* ***updateInt*** to update *String and int data respectively*

- Each of these methods takes two arguments:
  - A string specifying the name of the field to be updated;
  - A value of the appropriate type that is to be assigned to the field

- There are three steps involved in the process of **updating**:
  - position the ***ResultSet*** *cursor at the required row;*
  - call the appropriate ***update****** *method(s);*
  - call method ***updateRow***

```
results.absolute(2);   //Move to row 2 of ResultSet
results.updateFloat("balance", 42.55f);
results.updateRow();
```

Note here that an 'f' must be appended to the float literal, in order to prevent the compiler from interpreting the value as a double

- For an **insertion**, the new row is initially stored within a special buffer called the 'insertion row' and there are three steps involved in the process:
  - call method *moveToInsertRow;*
  - call the appropriate ***update**** method for each field in the row;*
  - call method *insertRow.*

```
results.moveToInsertRow();
results.updateInt("acctNum", 999999);
results.updateString("surname", "Harrison");
results.updateString("firstNames", "Christine Dawn");
results.updateFloat("balance", 2500f);
results.insertRow();
```

- ***Get**** *methods called after insertion will not retrieve* values for newly-inserted rows

- If this is the case with a particular database, then it will be necessary to close the ***ResultSet*** *and create a new one (using the original* query), in order for the new insertions to be recognized

- To delete a row without using SQL, there are just two steps:
  - move to the appropriate row
  - call method ***deleteRow***

Example:

```
results.absolute(3);        //Move to row 3.
results.deleteRow();
```

- Note that JDBC drivers can handle deletions differently

- Some remove the row completely from the *ResultSet,* while others use a blank row as a placeholder

- *With* the latter, the original row numbers are not changed

Thank You …